



下载APP

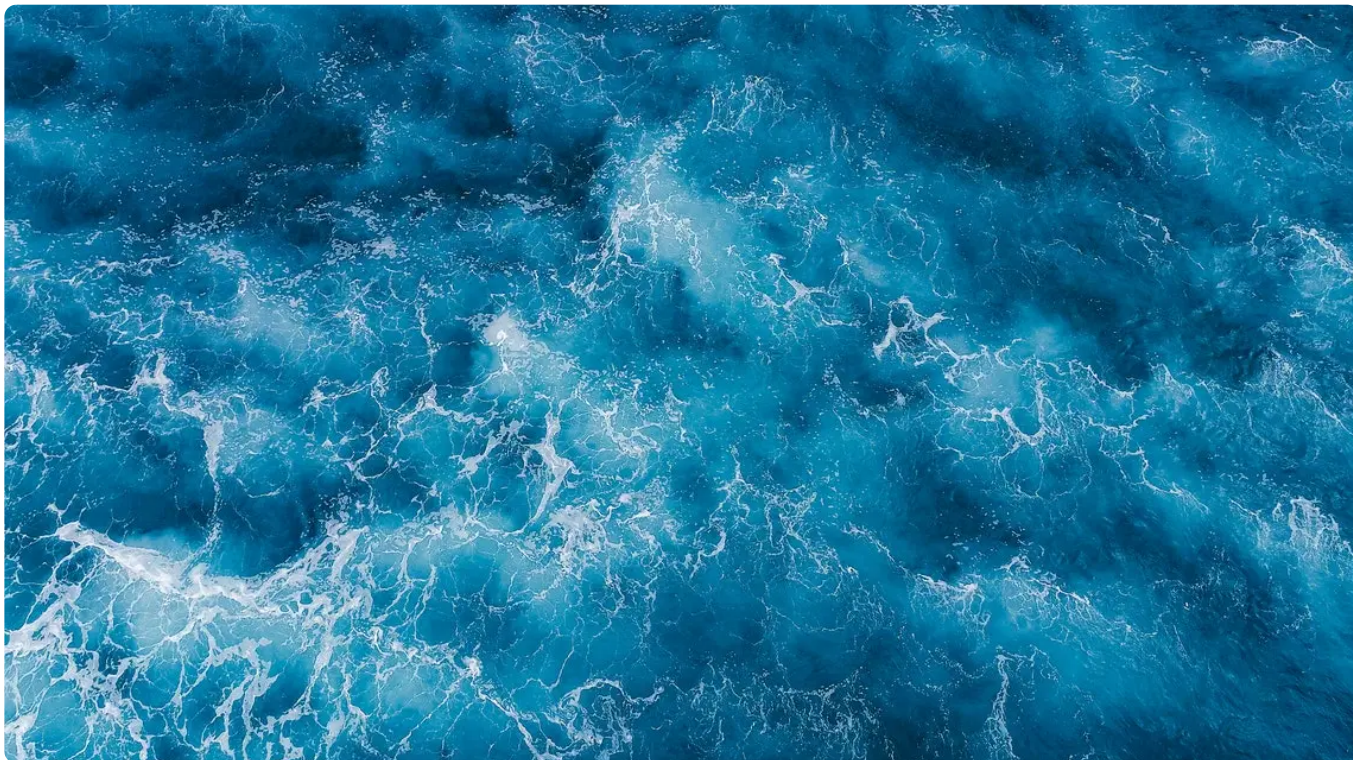


10 | 标准库：字符、字符串处理与数学计算

2021-12-29 于航

《深入C语言和程序运行原理》

课程介绍 >

**讲述：于航**

时长 13:14 大小 12.12M



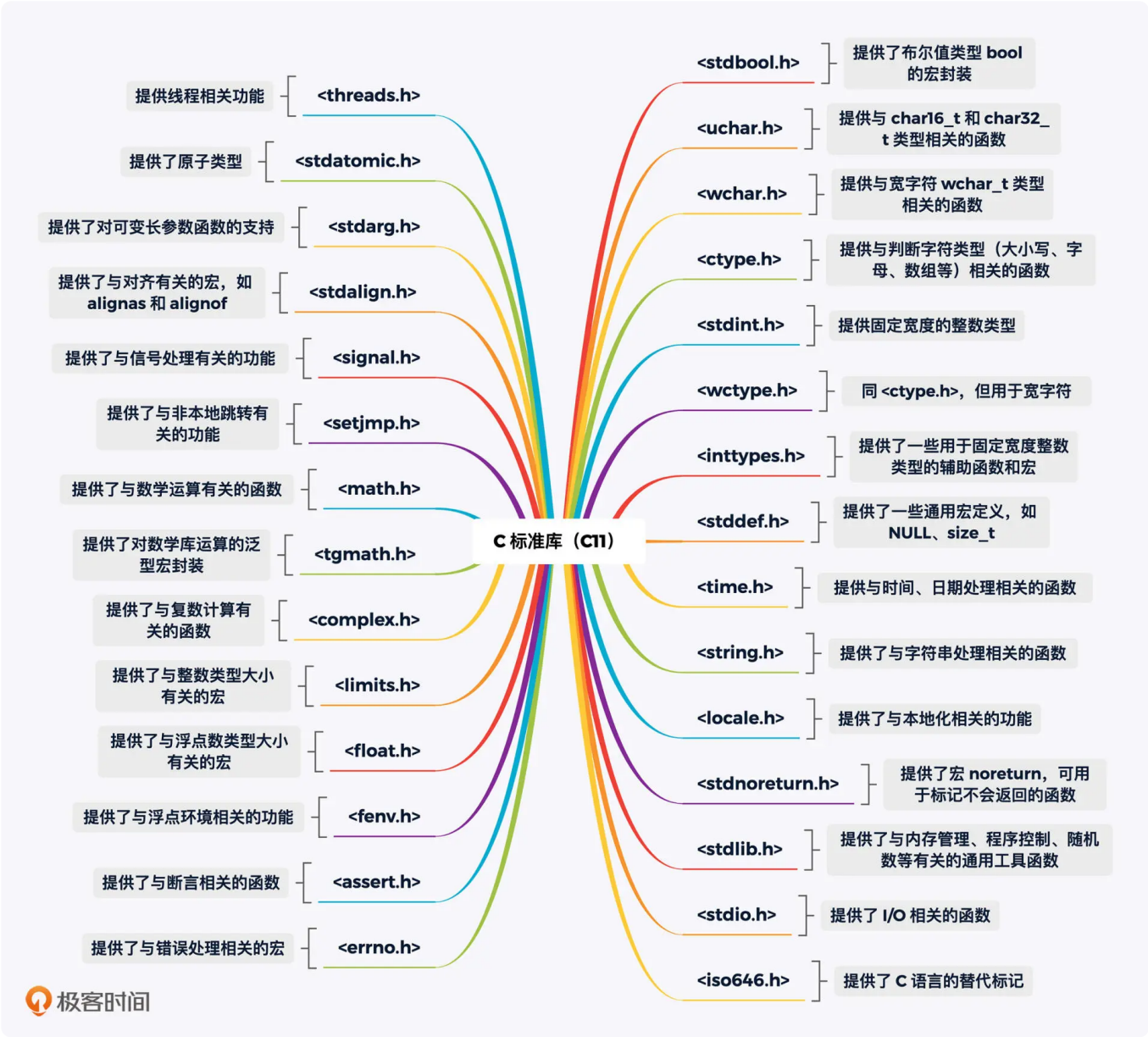
你好，我是于航。从这一讲开始，我们将进入到 C 工程实战篇的学习。

在上一个模块中，我主要围绕着 C 语言的七大关键语法，介绍了它们在机器指令层面的实现细节。而接下来，走出语法，从微观到宏观，我们将开始进一步探索 C 语法之外，那些可以用来支撑大型 C 项目构建的特性和技术。

领资料

因此，在这一模块中，我会介绍和 C 语言标准库、工程化，以及性能优化等相关的内容。C 标准库是除 C 基本核心语法外，C 语言的另一个重要组成部分。C 标准库中提供了众多的类型、函数和宏，可供我们直接在程序中使用。这些“构建单元”的功能涵盖了多方面，从简单的文本字符处理，到复杂的线程操作、内存管理等。在继续后面的内容之前，你可以先粗略浏览下图，以对 C 标准库提供的基本功能有个大致的印象。





今天，我们就先来看看 C 标准库中与字符、字符串处理，以及数学运算相关的内容。

C 语言中的字符和字符串

在 C 语言中，字符用单引号表示，字符串用双引号表示。比如在下面这段代码中便定义有两个变量，它们分别存放了一个字符类型和一个字符串类型的值。

复制代码

```
1 char c = 'a';
2 const char* str = "Hello, geek!";
```

下面，我们就来分别看看与这两种类型有关的重要特性。

字符

在 C 语言标准中，不同于其他整数类型（比如 `int`），字符类型 `char` 并没有被规定默认的符号性，而其具体符号性则是由编译器和所在平台决定的。虽然在大多数情况下，编译器会选择将默认情况下的 `char` 类型视为有符号整数类型，但考虑到程序的可用性与可移植性，在实际编码中还是建议**显式指定出所定义字符变量的符号性**。

C 标准中之所以这样规定，主要源于其历史原因。比较有代表性的一种说法是：C 语言在设计之初参考了它的“前辈”B 语言。B 语言于 1969 年由 Ken Thompson 在贝尔实验室开发。作为一种无类型语言，它的字符类型仅用于存放字符，而不作数学运算之用。因此，并不需要特别区分符号性。所以在 C 语言中，作者也没有为字符类型规定默认的符号性。

另外，有关字符类型的另一个重要特征是，C 语言保证 `char` 类型只占用一个字节大小，因此在使用 `sizeof` 运算符计算该类型大小时，将永远得到结果 1。但事实上，并不是所有计算机体系都使用 8 位的字节大小。对于某些较为古老的计算机体系，一个字节可能对应大于或小于 8 位。

对于某些特殊的业务需求和功能场景，你可以通过访问标准库中定义的常量 `CHAR_BIT`，来检查当前体系下一个字符类型所占用的位数，该常量的使用方式如下所示：

 复制代码

```
1 #include <limits.h>
2 #include <stdio.h>
3 int main(void) {
4     printf("char type has %lu byte.\n", sizeof(char)); // ...1.
5     printf("char type has %d bits.", CHAR_BIT); // ...8.
6 }
```

不过需要注意的是，自 C89 以来的标准中规定，`CHAR_BIT` 的值不能小于 8，因此对于单个字节数小于 8 位的体系架构，该常量并不能反映真实情况。

最后介绍一个编码方面的特性。C 语言中的 `char` 字符类型变量在按照字符类型打印时（比如使用 `printf` 函数，配合 “%c” 占位符），会使用 ASCII 编码来对数字值进行字符映射。这意味着，一个存储有整数 65 的字符类型变量，在将其打印输出时，会在命令行中

直接得到字符 “A” 。当然，你也需要确保命令行客户端的编码设置为 UTF-8，以与 ASCII 编码相兼容。

除了 char 类型以外，C 语言还在 C90 和 C11 标准中新增了可用于宽字符的类型，诸如 wchar_t、char16_t、char32_t 等。其中，wchar_t 类型的大小由具体的编译器实现决定。而 char16_t 和 char32_t 类型，其值则固定占用对应的 16 和 32 位。

字符串

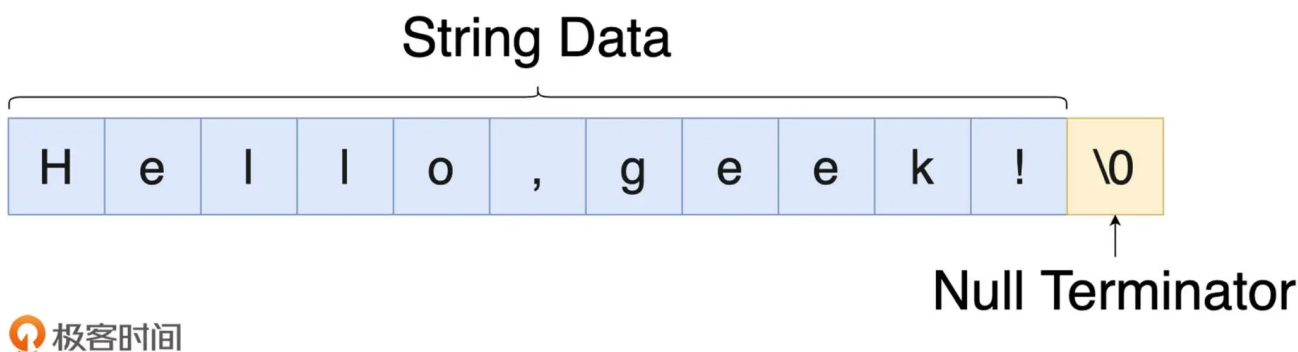
在 C 语言中，我们可以通过下面这两种方式来定义字符串类型的变量，一种是指针形式，另一种是数组形式。当然，这里示例代码中我们定义的是只读字符串：

 复制代码

```
1 // read-only string.
2 const char strA[] = "Hello, geek!";
3 const char* strB = "Hello ", "geek!";
```

其中，由双引号括起来的部分一般称为“字符串字面量”。C 标准中规定，连续出现的字符串字面量之间如果仅由空格分隔，则会将它们视为一个整体。所以 strA 与 strB 这两个字符串的内容是完全一样的。本质上，这两种方式都展示出了字符串在 C 语言中的具体表现形式，即“带有终止符的字符数组”。

关于上述字符串在内存中的布局形式，我们可以用下面这张图来直观地理解一下。可以看到的是，字符串数据被存放在连续的内存段上，且每个字符串数据的最后都以空字符（\0）作为表示结束的终止符。所以我们说，字符串并不是单纯的字符数组，而是带有隐式（定义时会被自动加上）终止符的字符数组。



虽然通过上面这两种方式都可以定义字符串，但实际上，**不同的定义方式在不同情况下可能会对程序运行产生不同的影响**。这里你可以先思考下：假设我们有如下这段代码，它在运行时会有什么问题吗？如果有，那应该怎样改正？

 复制代码

```
1 #include <string.h>
2 #include <stdio.h>
3 int main (void) {
4     /* get the first token */
5     const char* token = strtok("Hello, geek!", "geek");
6     printf("%s", token);
7     return 0;
8 }
```

上面这段代码在不同平台上运行可能会得到不同的结果。比如在 Linux 上，你可能会得到名为 “Segmentation fault” 的系统错误。如果进一步用 LLDB 进行调试，你会发现错误原因是：“signal SIGSEGV: address access protected”，翻译过来就是访问了受保护的地址。那接下来我们一起看看为什么会这样。

我们在之前的内容中曾提到过，字符串常量一般会被存放在进程 VAS 的 .rodata Section（下文简称 .rodata）中，位于此处的数据一般可以在程序中被多次引用。而当数据需要被修改，或以较短生命周期的形式（如局部变量）存在时，其引用的相关数据可能需要从 .rodata 中被复制到其他位置。而上述这两种字符串定义方式便对应于这两种情况。

以本小节开头的代码为例，使用指针形式定义的字符串 strB，实际上直接引用了 .rodata 中该字符串的所在位置，即字符指针 strB 的值为该位置对应的内存地址。而使用数组形式定义的字符串 strA，则是将该字符串的数据从 .rodata 中复制到了其他地方，strA 的值为复制后该字符串第一个字符的所在地址。

我们可以通过下面这段代码来验证这个结论。

 复制代码

```
1 // string.c
2 #include <stdio.h>
3 int main(void) {
4     const char strA[] = "Hello, geek!";
5     const char* strB = "Hello, geek!";
```

```

6  printf("%p\n%p", strA, strB);
7  /**
8   Output:
9   0x7ffee84d3d0b
10  0x4006c8
11  */
12 }

```

上面的代码中，我们使用 “%p” 格式符来打印变量 strA 与 strB 这两个指针的值。可以看到，当在 Linux 下执行这段代码时，变量 strA 与 strB 分别对应两个完全不同长度的地址（参考代码后的注释）。此时，我们可以通过如下命令来查看当前进程的 VAS 分布情况。

[复制代码](#)

```
1 pgrep string | xargs -I {} cat /proc/{}/maps
```

命令执行后，会得到如下图所示结果：

```

[root@iZ23ki8vt8tZ workspace]# pgrep string | xargs -I {} cat /proc/{}/maps
00400000-00401000 r-xp 00000000 ca:01 2911579 /www/workspace/string
00600000-00601000 r--p 00000000 ca:01 2911579 /www/workspace/string
00601000-00602000 rw-p 00001000 ca:01 2911579 /www/workspace/string
02208000-02229000 rw-p 00000000 00:00 0 [heap]
7f573bbd5000-7f573bd91000 r-xp 00000000 ca:01 50333416 /usr/lib64/libc-2.28.so
7f573bd91000-7f573bf90000 ---p 001bc000 ca:01 50333416 /usr/lib64/libc-2.28.so
7f573bf90000-7f573bf94000 r--p 001bb000 ca:01 50333416 /usr/lib64/libc-2.28.so
7f573bf94000-7f573bf96000 rw-p 001bf000 ca:01 50333416 /usr/lib64/libc-2.28.so
7f573bf96000-7f573bf9a000 rw-p 00000000 00:00 0
7f573bf9a000-7f573bfc6000 r-xp 00000000 ca:01 50333408 /usr/lib64/ld-2.28.so
7f573c1be000-7f573c1c0000 rw-p 00000000 00:00 0
7f573c1c6000-7f573c1c7000 r--p 0002c000 ca:01 50333408 /usr/lib64/ld-2.28.so
7f573c1c7000-7f573c1c9000 rw-p 0002d000 ca:01 50333408 /usr/lib64/ld-2.28.so
7ffee84b5000-7ffee84d6000 rw-p 00000000 00:00 0 [stack]
7ffee85f7000-7ffee85fb000 r--p 00000000 00:00 0 [vvar]
7ffee85fb000-7ffee85fd000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```

可以看到，以字符数组形式定义的字符串，其对应变量 strA 的数据实际上会从 .rodata 中被复制到当前进程 VAS 的栈内存中。而当程序运行脱离 strA 所在的作用域时，该数组对应的值将会被释放。反观以指针形式定义的字符串 strB，通过执行以下命令，我们也可以证实其指针所指向的位置为 .rodata。

[复制代码](#)

```
1 objdump -s string | grep .rodata -A 10
```

该命令会直接打印出当前程序 `.rodata` 的相关情况。可以看到，最右侧解码后的 ASCII 字符串 `"Hello, geek!"` 正对应着值为 `0x4006c8` 的起始地址。

```
[root@iZ23ki8vt8tZ workspace]# objdump -s string | grep .rodata -A 10
Contents of section .rodata:
 4006b8 01000200 00000000 00000000 00000000 .....
 4006c8 48656c6c 6f2c2067 65656b21 0025700a Hello, geek!.%p.
 4006d8 257000                                     %p.
```

最后我们总结一下。使用数组和指针形式定义的字符串，其底层的数据引用方式会有所区别。其中数组方式会将字符串数据从 `.rodata` 中拷贝到其他位置（比如栈内存），因此修改这些数据不会改变存在于原始 `.rodata` 中的副本。而使用常量指针形式定义的数组，该指针会直接引用位于 `.rodata` 中的字符串数据。

因此，我们需要注意的一个问题是：当使用非 `const` 指针引用字符串时，通过该指针修改字符串的值，可能会影响到其他使用指针形式引用相同字符串的代码位置上的实际字符串值。当然在 C 标准中，这种修改方式本身是一种未定义行为，其产生的具体影响将由编译器和操作系统决定。但大多数情况下，该行为都会产生诸如 `"Segmentation fault"` 以及 `"Bus error"` 之类的系统错误。

C 标准库中的字符、字符串处理

C 标准库中提供了众多的函数，可供我们直接对字符和字符串数据进行处理，这里我选择性地介绍其中的一些常见用例。对于这些 C 标准库函数的更详细的使用方法，你可以在 [这里](#) 查阅相关文档。

统计字符串长度

 复制代码

```
1 #include <string.h>
2 #include <stdio.h>
3 int main(void) {
4     const char str[10] = "Hi";
5     printf("%zu\n", strlen(str)); // 2.
6 }
```

这里我们直接使用标准库提供的 `strlen` 函数，**该函数不会计入字符串中多余的终止符**。

拼接字符串

[复制代码](#)

```
1 #include <string.h>
2 #include <stdio.h>
3 #define STRLEN 14
4 int main(void) {
5     char strA[STRLEN] = "Hello,";
6     char strB[] = " world!";
7     strncat(strA, strB, STRLEN - strlen(strA) - 1);
8     printf("%s\n", strA);
9 }
```

在这个例子中，我们选择使用 `strncat` 函数来进行字符串拼接。该函数相较于 `strcat` 函数，可以更好地控制被拼接字符串的长度，以防被拼接字符串过长导致的拼接字符串数组溢出。这里需要注意，在计算 `strncat` 函数的第三个参数，也就是被拼接字符串长度这个参数时，需要为字符串最后的终止符预留出 1 个字节的空間。

拷贝字符串

[复制代码](#)

```
1 #include <string.h>
2 #include <stdio.h>
3 int main(void) {
4     char strA[] = "aaaaaa";
5     char strB[] = "bbbbbbb";
6     printf("%s\n", strncpy(strA, strB, strlen(strA))); // "bbbbbb".
7 }
```

拷贝字符串函数 `strncpy` 的用法与 `strncat` 基本相同，我们可以控制其第三个参数，来决定将多少字符拷贝到目的字符串的数组中。这里我给你留下一个小问题：如果把 `strncpy` 函数中第三个参数使用的 `strlen` 函数更换成 `sizeof`，那么程序运行会得到什么结果？为什么？你可以在评论区和交流讨论。

格式化字符串

[复制代码](#)

```
1 #include <stdio.h>
```

```
2 #define LEN 128
3 int main(void) {
4     char dest[LEN];
5     const char strA[] = "Hello, ";
6     sprintf(dest, "%sworld!", strA);
7     printf("%s\n", dest);
8 }
```

函数 `sprintf` 可用于格式化字符串，其第二个参数的形式与 `printf` 函数的第一个参数一致，只是后者会将内容输出到命令行中，而 `sprintf` 会将格式化后的字符串保存到通过其第一个参数传入的数组中。

字符的判断与转换

在 C 标准库头文件 `ctype.h` 中包含有众多可用于字符判断和转换的函数，这些函数自身的名称直接说明了它们的具体功能，使用方式十分简单。具体你可以参考下面这个实例。

[复制代码](#)

```
1 #include <ctype.h>
2 #include <stdio.h>
3 int main(void) {
4     char c = 'a';
5     printf("%d\n", isalnum(c)); // 1.
6     printf("%d\n", isalpha(c)); // 1.
7     printf("%d\n", isblank(c)); // 0.
8     printf("%d\n", isdigit(c)); // 0.
9     printf("%c\n", toupper(c)); // 'A'.
10 }
```

C 标准库中的数学运算

上面我介绍了 C 标准库中与字符和字符串处理相关函数的使用方式，除此之外，C 标准库中还提供了与数学运算有关的工具函数，基本上你都可以通过引入 `math.h` 和 `stdlib.h` 这两个头文件来使用。这些函数的使用方式都十分简单，你可以在 [这里](#) 找到对它们用法的详细说明。不过有一点要注意：在编译时，你可能需要为链接器指定 “`-lm`” 参数以链接所需的数学库。

在下面这个简单的例子中，我们使用了标准库中的求绝对值函数。

 复制代码

```
1 #include <math.h>
2 #include <stdio.h>
3 int main(void) {
4     long double num = -10.1;
5     printf("%.1Lf\n", fabsl(num));
6 }
```

为了减少编码的工作量，我们也可以使用这些函数对应的泛型版本，这样就不需要根据传入的参数来手动选择合适的版本（比如这里的 `fabsl` 函数，它的后缀 “l” 表示 “long”）。比如，对于所有浮点类型的数字值，我们可以直接使用名为 `fabs` 的宏函数。该宏在展开时，会自动为你匹配对应的类型精确版本。不过，为了使用这些泛型宏，我们需要将原来的头文件 `math.h` 替换为 `tgmath.h`，如下代码所示：

 复制代码

```
1 #include <tgmath.h>
2 #include <stdio.h>
3 int main(void) {
4     long double num = -10.1;
5     printf("%.1Lf\n", fabs(num));
6 }
```

关于这些标准库函数的实现方式，如果我们进一步来看，会发现并非所有函数都是按照相应的数学算法来实现计算过程的。

这里用一个常见的数学运算“求平方根”举例子：通常来说，我们可以使用牛顿迭代法，以软件算法的形式计算一个数的平方根值。但实际上，当我们以 `musl` 这个 C 标准库为例，进一步查看其某版本的实现时，可以看到它在 `i386` 架构下直接使用了 `FPU`（浮点运算单元）提供的机器指令 `fsqrt` 来计算平方根的值，而并没有使用软件算法。在某种程度上，这可以极大提升计算性能。如下图所示：

```
1  #include "libm.h"
2
3  double sqrt(double x)
4  {
5      union ldshape ux;
6      unsigned fpsr;
7      __asm__ "fsqrt; fnstsw %%ax": "=t"(ux.f), "=a"(fpsr) : "0"(x);
8      if ((ux.i.m & 0x7ff) != 0x400)
9          return (double)ux.f;
10     /* Rounding to double would have encountered an exact halfway case.
11        Adjust mantissa downwards if fsqrt rounded up, else upwards.
12        (result of fsqrt could not have been exact) */
13     ux.i.m ^= (fpsr & 0x200) + 0x300;
14     return (double)ux.f;
15 }
```

因此，为了最大程度地利用硬件带来的计算性能优势，在准备自行编写相应算法时，可以先看看能否直接利用标准库提供的函数来完成你的工作。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

今天我主要介绍了 C 语言标准库中与字符、字符串处理，以及数学运算有关的内容。

首先，我介绍了 C 语言中字符和字符串类型变量的定义方式。字符以单引号形式定义，而字符串以双引号形式定义。字符串的不同定义方式还可能对程序的运行细节带来影响。其中，以字符数组形式定义的字符串包含有原始字符串在 .rodata 中的拷贝；而以指针形式定义的字符串变量则直接引用了 .rodata 中的字符串数据，且其值通常无法在程序运行时被动态修改。

接下来，我快速介绍了 C 标准库中与字符、字符串处理相关的一些函数的使用方式。最后，我介绍了 C 标准库中与数学运算相关的函数，math.h 头文件中包含有这些函数的精确类型版本，而 tgmath.h 头文件中则提供了对应的泛型宏函数版本。这些函数的一个重要特征是：某些常用的数学运算会被直接映射到对应的机器指令，和使用纯软件算法的实现相比，这通常可以获得更高的性能。

思考题

请了解一下 C 标准库中 strtok 函数的实现，并思考是哪一步引起了文中实例的

“Segmentation fault” 错误。然后，如果时间充足，可以尝试实现一个自己的版本，拥有与 strtok 函数一样的功能，但是不修改传入的源字符串。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | 编译准备：预处理器是怎样处理程序代码的？

下一篇 课堂答疑（一） | 前置篇、C 核心语法实现篇问题集锦

更多课程推荐

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

陈天

Tubi TV 研发副总裁



涨价倒计时 

今日订阅 **¥89**，1月12日涨价至**¥199**

精选留言 (8)

 写留言



Fan



2021-12-29

不错，就喜欢这种讲细节的课程。

**白花风信子**

2021-12-29

学到好多细节！不过老师会在后面再深入提提elf文件是怎么加载的嘛qwq。

作者回复：会的哈，我们会在第三个模块中介绍的。

**赵岩松**

2021-12-30

终于到了我期待的讲标准库的环节

**liu_liu**

2021-12-31

strtok 的原型如下：

```
char *strtok(char *src, char *delim) ; ...
```

展开 ▾

**ppm**

2021-12-29

写了一个strtok <https://gist.github.com/goog/f41b7b7fcdb7e9de93606c370c9d22b5>

**blentle**

2021-12-29

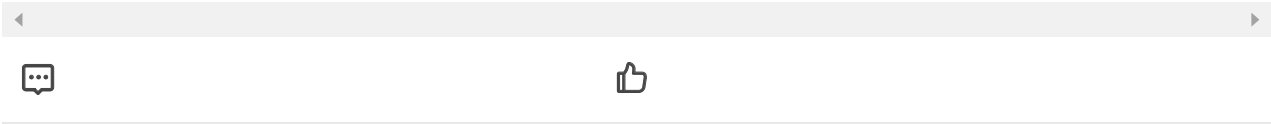
学到了.

**Ping**

2021-12-29

老师能不能介绍一下musl这个库的特点和用途？

作者回复: musl 其实跟我们常用的 glibc (GNU C Library) 类似，两者同属于 C 标准库 (libc) 的一种实现。只是相对来说，glibc 实现较为臃肿，健壮性稍差，存在的潜在漏洞也多。但是实现的功能众多（有很多 C 标准以外的），且使用也很广泛。而 musl 则旨在提供一个轻量级的、ABI 稳定的、且适用于移动设备和嵌入式系统的 libc 实现。它支持的功能上则对标 C 标准。



ppm
2021-12-29

怎么没有snprintf的例子

作者回复: 嗯，查了一下才发现 snprintf 比 sprintf 函数更安全一些。新知识很到位！

